NIST
National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

The OXCSwitch in GMPLS Lightwave Agile Switching Simulator (GLASS)

Version: Draft 1.0

TABLE OF CONTENTS

1 INTRODUCTION 1

2 CAPABILITIES 2

2.1 THE SWITCHING TABLE 3
2.2 CONVERTERS 3
2.3 CONCATENATION 4
2.4 ADD DROP CAPABILITY 4

3 CONFIGURATION 5

4 IMPLEMENTATION 6

4.1 UML DIAGRAM 6
4.2 HOW TO USE THE OXCSWITCH 7
4.2.1 From an algorithm 7
4.2.1.1 Converters 7
4.2.1.2 Concatenation 7
4.2.1.3 Connection 8
4.2.2 From a protocol 9
4.2.2.1 Add/Drop information 9
4.2.2.2 Add/Drop configuration 10
4.2.2.3 Sending/Receiving messages 12

5 CONCLUSION 13

6 ANNEX 13

1 INTRODUCTION

This document presents the optical switch that is provided in the GLASS Framework. The **OXCSwitch** is an SSFNet protocol session but in GLASS the switch belongs to the host in lieu of the protocols. The OXCSwitch configuration changes the behavior of the whole Optical Cross Connect (OXC). That is the reason why it is important to know the effects of changing some of its attributes in the configuration.

The next section provides information about the global capabilities of the OXCSwitch. Then we go in the configuration followed by the implementation and how to use the OXCSwitch with a new protocol.

It is recommended to look into the coding of the classes mentioned in this document for implementation details. A list of the classes is attached to the Annex.

2 CAPABILITIES

The default implementation of the optical switch is available in the package **gov.nist.antd.optical** and implemented in the class **OXCSwitch**. This class provides a configurable switch that includes the main features of a real optical switch.

Figure 1 displays the position of the optical switch in the design of a GLASS OXC node.

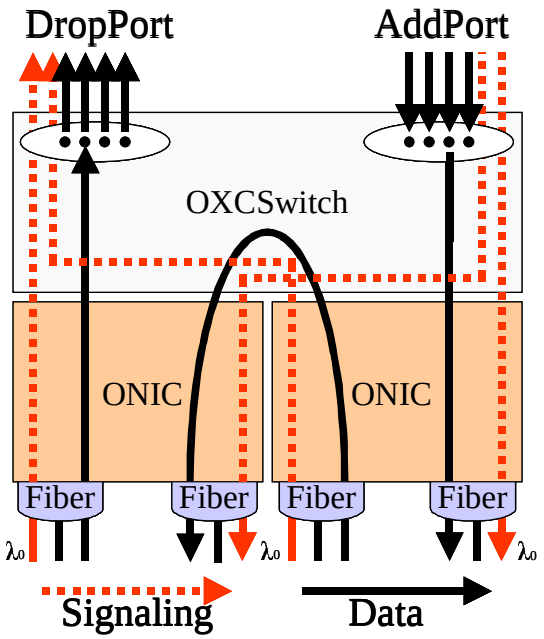

Figure 1: The OXCSwitch

The *OXCSwitch* is located on top on the optical network interface cards (ONIC) and below the other protocols like IP (if the implementation of IP knows the optical framework).

The switch is used to connect input lambdas to output lambdas for direct O/O/O[1] switching or to upper layers for O/E/O[2] switching or just as Add/Drop capability. To do this, the switch contains a switching table that can be dynamically updated before and during the simulation by a standardized interface.

[1] optical input, optical switching, optical output

[2] optical input, electronical switching, optical output

In the GLASS framework, the switch opens the ONICs below it to receive the packets from the network.

The switch is the last optical layer. When a message is received by a ONIC and is forwarded to another ONIC without going to some upper layer (O/O/O switching), it stays in the optical domain. If the switch pushes up to another protocol then there is a conversion from optical to electronic.

2.1 THE SWITCHING TABLE

The switching table connects two lambdas. In fact, these lambdas can be either physical lambdas (located in the fiber of a link), or logical lambdas (for add/drop capabilities).

The current version does not provide any multiplexing in the switching table. This means it is a one to one relation. The multiplexing can be done on the layers on top of the OXCSwitch. The manipulation of the switching table is possible only indirectly and has to be done through the OXCSwitch that provides mechanisms to ensure the integrity of the table.

2.2 CONVERTERS

The behavior of the switch, or the possibilities of the switch is on one hand due to the converters. The converter attribute specifies the number of available converters and therefore indicates the capability of lambda switching.

The lambda conversion allows the switch to change the wavelength while switching. For example an input lambda of 1550nm can be connected to an output lambda of 1551.6nm. When the connect method is called in the switch, then it looks if the values are correct. For example, if the switch does not provide lambda conversion (because there was not converters or because they are all used), then trying to realize the switching given before, will produce an exception.

2.3 CONCATENATION

The concatenation of the OXCSwitch is the capability to treat a set of input lambda a same way also called waveband switching. The current implementation allows three configurations:

- No concatenation: If the switch is in the configuration mode then no waveband switching is possible.
- Standard concatenation: The wavelengths that compose a waveband must be contiguous.
- Virtual concatenation: The wavelengths need not to be contiguous.

Currently this is only a characteristic of the switch but no checking has been implemented to control its behavior depending on the configuration.

2.4 ADD DROP CAPABILITY

The OXCSwitch does not only provide O/O/O switching. It is also possible to transmit information coming from an optical link to protocols. In this case, the switch needs to convert the signal to electronic. In general, using the so-called “add-drop-lambdas” does the communication between the protocols and the switch. Chapter 2.4 gives more explanation on how to use them. These add/Drop lambdas (ADL) are mapped one by one with a physical lambda. An ADL can be connected to an input or to an output lambda. The implementation of an ADL is in the class AddDropLambda.

3 CONFIGURATION

This section contains the DML schema of the OXCSwitch and explains how to configure it.

ProtocolSession [
  name oxcwitch
  use gov.nist.antd.optical.OXCSwitch
  noConverters %I
  concatenation %S
  noAddDrop %I
]
	Optional attribute noConverters indicates the number of converters included in switch (default value = infinite).
	
	Optional attribute concatenation defines the type of concatenation available for waveband switching. Three types are defined: none, standard (lambdas must be contiguous), virtual (lambdas don't necessary need to be contiguous).
	
	Optional attribute noAddDrop specifies the number of AddDropPort in the switch. These ports are used to connect other protocols on top of the OXCSwitch (default value: max between the inLambdas and outLambdas).

Table 1: The OXCSwitch Configuration Schema

If the user does not enter specification to the OXCSwitch, then the result is:

- Unlimited converters so that there is no restriction for the wavelength algorithms.
- No concatenation which means that there is no waveband switching.
- A number of add/drop lambda equals to the number of physical lambdas to allow all possible combinations to switch lambdas.

4 IMPLEMENTATION

This section introduces the reader to the design of the class **gov.nist.antd.optical.OXCSwitch** and the related classes.

4.1 UML DIAGRAM

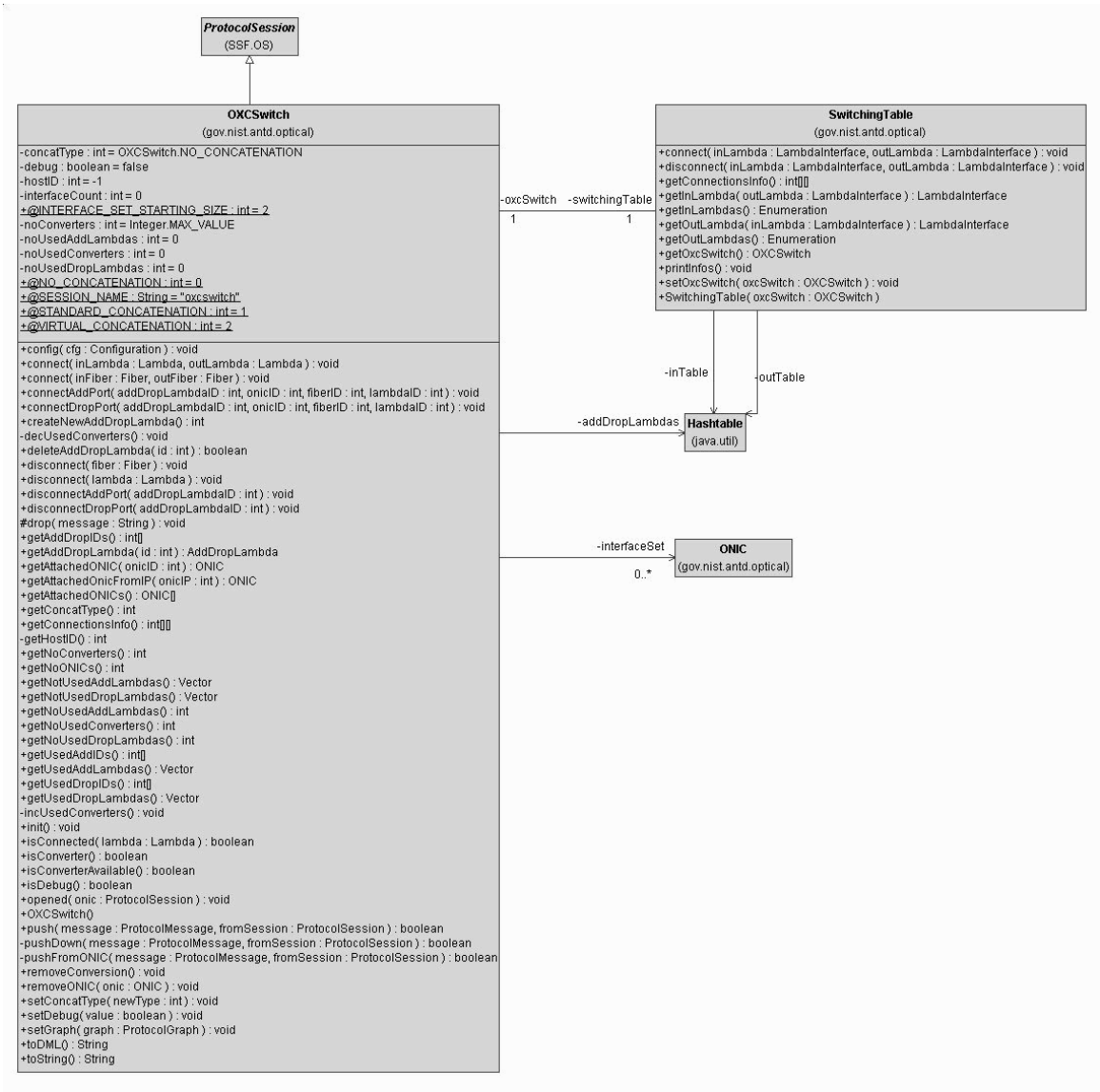

Figure 2: UML diagram of OXCSwitch

4.2 HOW TO USE THE OXCSWITCH

4.2.1 FROM AN ALGORITHM

The routing and wavelength algorithms may be interested in some characteristics of the switch to compute the route or path. This paragraph shows how to get the information.

4.2.1.1 CONVERTERS

To get the information about converter, 2 methods must be used:

- *public boolean isConverter()*: Returns true is the switch contains converters.
- *public boolean isConverterAvailable()*: Returns true if there is at least one converter available.

The maintenance of the number of converters is done automatically by the OXCSwitch every time the method *connect ()* or *disconnect ()* is called. This information is very important for a wavelength algorithm to decide if this node can be used or not. The switch will reject any connection between two lambdas with different wavelength if there is no converter available.

4.2.1.2 CONCATENATION

The OXCSwitch class contains some static constant to identify the concatenation type:

- NO_CONCATENATION
- STANDARD_CONCATENATION
- VIRTUAL_CONCATENATION

Chapter 2.3 explains for the signification of these values in detail.

To access the concatenation-status of the switch, the following method is available:

- *public int getConcatType( )* : Returns the value of the concatenation type.

This concatenation type is set during the configuration. It is also possible to modify the value even during the simulation by using the corresponding setter method.

4.2.1.3 CONNECTION

A wavelength algorithm may want to connect two single lambdas or two complete fibers. For this purpose, the OXCSwitch provides 2 methods:

- *public void connect(Lambda inLambda, Lambda outLambda)*:Connects the 2 given lambdas. An exception may occur for different reasons. For example, if a lambda is already connected. It happens also if the lambdas have different wavelength and the switch has no more converter available.
- *public void connect(Fiber inFiber, Fiber outFiber)* : This method connects all the lambdas of the input fiber to the lambdas of the output fiber. This method provides only a basic checking mechanism. The fibers must have the same number of lambdas and the switch tries to connect the lambda without any optimization. It tries to connect the first lambda of the input fiber to the first lambda of the output fiber and so on.

It may be possible that the implementation of an algorithm needs more information that is explained in the following Chapter. Also a protocol may want to connect two lambdas. The way to access information is the same for an algorithm or a protocol.

4.2.2 FROM A PROTOCOL

When a protocol wants to use the switch to transmit data or to manipulate the configuration, it needs to request the status of the switch first, to prevent any kind of exception or conflict.

4.2.2.1 ADD/DROP INFORMATION

The add/drop information must be used when a protocol needs to send data using the optical switch. The following methods can be used to determine the availability of the resources:

- *public int getNoUsedAddLambdas( )*: Returns the number of virtual lambdas at the add port.
- *public int getNoUsedDropLambdas( )*: Returns the number of virtual lambdas at the drop port.
- *public Vector getNotUsedAddLambdas( )*: Returns a vector that contains the virtual lambdas available at the add port. If one is available, then a protocol can use it to send information to the network through a physical lambda.
- *public Vector getNotUsedDropLambdas( )*:Returns a vector that contains the virtual lambdas available at the drop port. If one is available, a protocol can use it to receive information from the network through a physical lambda.
- *public int[] getUsedAddIDs( )*: Returns an array that contains the id of the virtual lambdas used at the add port.
- *public int[] getUsedDropIDs( )*: Returns an array that contains the id of the virtual lambdas used at the drop port.
- *public Vector getUsedAddLambdas( )*: Returns a Vector that contains the virtual lambdas used at the add port.
- *public Vector getUsedDropLambdas( )*: Returns a Vector that contains the virtual lambdas used at the drop port.

Once a protocol sees that there are resources available, it can connect the ADL to the physical lambdas using the following methods:

- *public void connectAddPort (int addDropLambdaID, int onicID, int fiberID, int lambdaID)*
  This method connects the virtual lambda addDropLambdaID of the add port with the lambda identified by the onicID, fiberID and lambdaID. If the connection is done, a protocol will be able to send data to the network. Otherwise an InvalidConnectionException is thrown if the switch finds an error.
- *public void connectDropPort (int addDropLambdaID, int onicID, int fiberID, int lambdaID)*
  This method connects the virtual lambda addDropLambdaID of the drop port with the lambda identified by the onicID, fiberID and lambdaID. If the connection is done, a protocol will be able to send data to the network. Otherwise an InvalidConnectionException is thrown if the switch finds an error.

There is opposite methods to disconnect an add port and a drop port given the port number.

4.2.2.2 ADD/DROP CONFIGURATION

This paragraph explains two other important features that help a protocol user to configure the add/drop ports. Some protocols, like signaling protocols, must be connected to add/drop ports in order to receive and send information.

The abstract class **gov.nist.antd.merlin.AbstractAddDropConfigurator** provides the class that extends it the possibility to configure each add/drop port out of the DML.

The format is as follow:

addLambda [
  Id $I1!
  onicId $I1!
  fiberID $I1!
  lambdaID $I1!
]
	Defines a connection of a lambda to an add port.

dropLambda [
  Id $I1!
  onicId $I1!
  fiberID $I1!
  lambdaID $I1!
]
	Defines a connection of a lambda to a drop port.

Table 2: The Add/Drop Lambda Configuration

When creating an add-lambda, the protocol will create a connection between the port and the lambda, and store the information of this add-port used to send data. Later on, the protocol can access this port without the need of manipulating the OXCSwitch. The same mechanism is used for the drop-port.
Checking mechanisms are also included and report exceptions if the lambda is unknown or if there are not enough ports available.

Another feature provides the user with a default connection setup of the add/drop ports without having to specify the configuration in the DML. The implementation is available in the class **gov.nist.antd.merlin.util.AutoCtrlConfig**.
This abstract class looks into each fiber connected to the OXCSwitch and connects the control lambdas. All control lambdas of output fibers are connected to add-ports (checks if there are enough ports available) and control-lambdas for input fibers are connected to drop-ports. For bidirectional fibers, the protocol will connect the control lambdas in both directions by alternations.

To use these capabilities, the protocol must extend one of these classes. The class AutoCtrlConfig extends the AbstractAddDropConfigurator and uses the same mechanisms to store the information about the add/drop port.

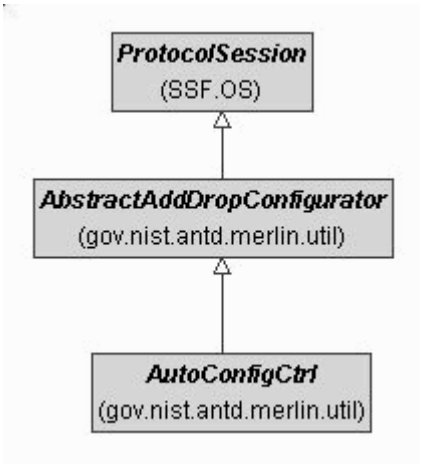

Figure 2: AbstractAddDropConfigurator and AutoconfigCtrl

The main objective of these classes is to provide a protocol designer with an example on how to create a protocol that connects itself to the OXCSwitch for sending/receiving data. The next

paragraph shows how the protocol, which is connected to the switch via an add/drop port, will send and receive messages.

4.2.2.3 SENDING/RECEIVING MESSAGES

For sending messages, the OXCSwitch uses the standard method of all ProtocolSession:

- *public boolean push(ProtocolMessage message, ProtocolSession fromSession)*

The protocol that needs to send messages is required to call this method to transmit data to the network. As the OXCSwitch is in the optical domain, the message type must be of OpticalFrameHeader.

The optical frame header (**gov.nist.antd.optical.OpticalFrameHeader**) is used for two reasons:

- To identify a lambda when a message is sent through an optical link.
  In SSFNet, a link represents only one cable so there is no problem to see where a message comes from. It is more complicated for an optical link that may contain multiple fibers and lambdas. That's why we use this header that contains an id for the fiber and the lambda.
- To identify which add-drop-port has to be used for sending or receiving a message.
  When a protocol wants to send a message, it has to create an instance of OpticalFrameHeader and specify the add-port to use in the field LambdaID of this header (the attribute fiberID must not be used). Then the switch looks into this header to find the corresponding lambda it has to send the message to. On the other direction, the switch knows (out of the header) from which physical lambda the message came from and updates the fields of the header. It replaces the LambdaID by the drop-port ID and set the attribute FiberID to "–1". Then it sends it to the protocol that is attached to this drop-port. The protocol, that received this message is able by checking at the header, to recognize from which port the message originally came from and processes it.

5 CONCLUSION

As explained in this document, simulation results may vary a lot, depending on the configuration of the switch. The modification of the converters and the add-drop-ports can influence the results of the algorithms. Even so some connections may fail if there are not enough resources free that are specified in the OXCSwitch.

The OXCSwitch is still under development and some improvements are under study. For example, the multiplexing/demultiplexing in the lambdas and switching delays.

6 ANNEX

This chapter contains the list of the class name mentioned in this document.

gov.nist.antd.optical.OXCSwitch

gov.nist.antd.optical.AddDropLambda

gov.nist.antd.optical.OpticalLink

gov.nist.antd.optical.Fiber

gov.nist.antd.optical.Lambda

gov.nist.antd.optical.InvalidConnectionException

gov.nist.antd.optical.OpticalFrameHeader

gov.nist.antd.optical.ONIC

gov.nist.antd.merlin.util.AbstractAddDropConfigurator

gov.nist.antd.merlin.util.AutoConfigCtrl

SSF.OS.ProtocolSession

SSF.OS.ProtocolMessage

SSF.Net.NIC

SSF.Net._NIC